

GenericRest. Version 1.0.0

Autor: Andrés García Meroño

Fecha: 20/10/2016

Índice de contenido


Introducción.....	1
Alta de un servicio.....	2
Servicios que despliega.....	4
Tipos de servicios.....	4
Operaciones del servicio.....	4
@GET ("/:table").....	4
@GET ("/:table/count").....	5
@GET ("/:table/:id(.*)").....	5
@POST ("/:table").....	5
@DELETE ("/:table/:id(.*)").....	6
@PUT ("/:table/:id(.*)").....	6
Otros servicios adicionales.....	6
@GET ("/mapperinfotable/:table").....	6
@GET ("/mapperinfolist").....	7
@GET ("/_inituser").....	7
@GET ("/_getuser").....	7
Sintaxis de fields.....	7
Sintaxis de filtro.....	8
Sintaxis de SEC_INFO.....	9
Sintaxis de mapperInfo dentro del fichero Spring para info y secInfo.....	9
Estructura de clases.....	10
Acerca de los Resolvers.....	11
Acerca de los SecurityResolvers.....	11


Introducción

Este documento explica de forma básica el funcionamiento y configuración de GenericRest.

Alta de un servicio

La pantalla de un servicio facilita su mantenimiento. Desde aquí se define el servicio REST, su comportamiento y la seguridad de acceso al servicio como de los datos a los que se puede acceder.

 **SERVICIOS**

 **Recarga de los servicios**

Previsualización de datos

Datos del servicio

←

+

TABLE_NAME

CARGO

RESOLVER

FIELDS

*

NEW TABLE_NAME

CARGO

FINAL_TABLE

KEYS

ID

SEPARATOR

SELECT_FILTER

Filtro dinámico para la consulta

SELECT_VALUE

Consulta de datos

INSERT_VALUE

Inserción de datos

UPDATE_VALUE

Actualización de datos

DELETE_VALUE

Borrado de datos

SEC_RESOLVER

Clase que implementa la seguridad

SEC_INFO

SIUDE=ADMIN Configuración de la seguridad

Metadatos del servicio

```
{
  "type": "null",
  "table": "CARGO",
  "keys": [
    "ID",
    ""
  ],
  "keySeparator": "/",
  "selectFilter": "null",
  "fields": [
    {
      "name": "ID",
      "type": "N",
      "size": "39",
      "sequenceName": "",
      "description": "ID"
    },
    {
      "name": "NOMBRE",
      "type": "T",
      "size": "50",
      "sequenceName": "",
      "description": "NOMBRE"
    },
    {
      "name": "DESCRIPCION",
      "type": "T",
      "size": "150",
      "sequenceName": "",
      "description": "DESCRIPCION"
    }
  ],
  "security": [
    {
      "key": "D",
      "roles": "ADMIN,"
    },
    {
      "key": "U",
      "roles": "ADMIN,"
    },
    {
      "key": "E",
      "roles": "ADMIN,"
    },
    {
      "key": "S",
      "roles": "ADMIN,"
    },
    {
      "key": "I",
      "roles": "ADMIN,"
    }
  ],
  "$promise": {},
  "$resolved": true
}
```

A continuación se describe cada uno de los campos:

- **TABLE_NAME:** nombre público del servicio. Es la etiqueta con la que se identifica cada servicio.
- **NEW_TABLE_NAME:** (obligatorio) permite cambiar el nombre del servicio.
- **FINAL_TABLE:** para servicios CRUD básicos de una tabla, es el nombre real de la tabla que se está maepando. Si se deja en blanco equivale a **TABLE_NAME**.
- **TYPE:** tipo de servicio. Dependiendo de este valor el servicio se comporta como un CRUD (campo en blanco), una llamada a una función (valor **FUNCTION**) o una llamada a un procedimiento (**PROCEDURE**). Este campo afecta a los métodos que expone el servicio.
- **FIELDS:** (obligatorio) lista de campos separada por comas de la definición de los campos del servicio. Permite definir también el tipo de dato y su longitud y cómo se intercambia la información desde o hacia el servicio. Ver apartado "Sintaxis de fields".
- **KEYS:** (obligatorio) lista de campos separada por comas de los que identifican la clave primaria de un registro.
- **SEPARATOR:** carácter que se usa para separar los diferentes campos de la clave primaria al inoar el servicio REST. Por defecto es '/'. Útil en aquellos casos donde algún campo de la clave primaria pueda contener el carácter '/'.
- **RESOLVER:** clase java que implementa cómo acceder a los datos. Por defecto es el `defaultResolver` de la configuración Spring.
- **SEC_RESOLVER:** clase java que implementa la seguridad de acceso a los datos. Por defecto es el `defaultSecResolver` de la configuración Spring.
- **SEC_INFO:** contiene la definición de la seguridad. Ver sintaxis **SEC_INFO**.
- **SELECT_VALUE, INSERT_VALUE, UPDATE_VALUE, DELETE_VALUE:** sobrescribe la implementación por defecto del resolver. Pudiendo escribir las diferentes cláusulas para un servicio tipo CRUD, o la llamada al un PL/SQL para un servicio de tipo procedure o function.

SELECT_VALUE se comprueba al arrancar el servidor o al recargal los servicios para verificar que la definición del campo **FIELDS** es compatible y no se ha cometido un error sintáctico. Esto implica que la definición de **SELECT_VALUE** es estática y no depende del usuario que la ejecuta. Por lo que no es aconsejable utilizar la sintaxis de Mybatis aquí.

En el caso de un procedure o función **SELECT_VALUE** es el campo donde se almacena el código PL/SQL y no se realiza comprobación alguna, por lo que el servicio puede fallar en tiempo de ejecución. El resto de campos no tienen asociada ninguna funcionalidad y deberían estar en blanco.

- **SELECT_FILTER:** Permite modificar la cláusula **SELECT_VALUE** de un servicio tipo CRUD con datos dinámicos en tiempo de ejecución, es decir, permite modificar los datos que devuelve el servicio en función del usuario que lo ejecuta. Aquí es normal utilizar la sintaxis de Mybatis para acceder a los metadatos del usuario.

Servicios que despliega

Hay tres tipos de servicios: CRUD, PROCEDURE y FUNCTION, dependiendo del tipo de servicio la configuración REST variará y las operaciones que muestra son diferentes.

Tipos de servicios

- **CRUD:** Este se caracteriza porque no tiene tipo, es el tipo por defecto. Despliega un servicio REST con las operaciones de crear, ver, editar y borrar.
- **PROCEDURE:** El valor del campo TYPE es "PROCEDURE" permite la ejecución de un código PL/SQL, normalmente una llamada a un procedimiento. Este tipo de operación devuelve un valor.
- **FUNCTION:** El valor del campo TYPE es "FUNCTION" permite la ejecución de un código PL/SQL, normalmente la llamada a una función, el resultado se pasa a la salida sin procesar.

Operaciones del servicio

@GET ("/:table")

Consulta de datos con selección de campos, paginación, ordenación, búsqueda full-text, búsqueda avanzada y formato de los datos devueltos.

PathParam:

- table: nombre del servicio, que es el valor del campo TABLE_NAME.

QueryParams:

- filter: permite seleccionar aquellos registros que cumplan una condición. Ver sintaxis de filtro. Por defecto no hay.
- limit: cantidad de registros que devuelve el servicio, por defecto 30.
- offset: a partir de qué registro devuelve los datos. Por defecto 0.
- orderby: lista de columnas separada por comas de los campos a ordenar.
- order: lista de sentido de ordenación separada por comas (sólo ASC y DESC).
- fields: lista de campos separada por comas o '*' para todos. Por defecto '*'.
- format: formato de salida de los datos (JSON, XLS, CSV, HTML, TXT). Por defecto JSON.

ReturnData: dependiendo del valor de format cambia el content-type:

- JSON: un array con el resultSet asociado a la consulta.
- XLS: los datos vienen en formato excel.
- CSV: los datos vienen separados por comas, una linea por registro.

- HTML: los datos vienen formateados usando TABLE, TR, TD en html.
- TXT: los datos vienen en texto separados por comas.

@GET (":table/count")

Devuelve la cantidad de registros que cumplen la condición. Se le debe pasar los mismos valores que el servicio de consulta para que el resultado sea coherente con los datos mostrados.

PathParam:

- table: nombre del servicio, que el valor del campo TABLE_NAME.

QueryParam:

- filter: igual que en el servicio de consulta.

ReturnData: un JSON tipo object con una propiedad "count" con el resultado. (ej: { count: 12345 })

@GET (":table/:id(.*)")

Devuelve un registro concreto en función de su clave primaria.

PathParam:

- table: nombre del servicio.
- id: lista de valores separados por el campo SEPARADOR uno por cada valor de la clave primaria en el orden en el que se definió en el campo KEYS.

ReturnData: un JSON tipo object con el registro asociado a la clave primaria.

Si la consulta devuelve más de un valor retorna un error.

El contenido de la clave primaria no puede contener el campo SEPARADOR, por ello hay que seleccionar otro valor si una campo de la clave primaria puede contener el caracter de SEPARADOR.

@POST (":table")

Realiza una inserción de un nuevo registro o la ejecución de una operación PL/SQL.

PathParam:

- table: nombre del servicio.

Body: objeto JSON con los datos del nuevo registro.

ResultData: en caso de servicios tipo CRUD devuelve el mismo objeto JSON que se pasa en el body. En otro caso el valor que devuelve la llamada PL/SQL.

@DELETE ("/:table/:id(.*)")

Borra un registro de datos según su clave primaria.

PathParam:

- table: nombre del servicio.
- id: lista de valores separados por el campo SEPARATOR uno por cada valor de la clave primaria en el orden en el que se definió en el campo KEYS.

ReturnData: Este servicio sólo responde un código de estado 200.

NOTA: aquí la lista de ids puede ser incompleta o **NULA**, lo que permite eliminar elementos que comparten parte de la clave primaria o incluso **TODOS** los datos si no se aporta id's.

@PUT ("/:table/:id(.*)")

Actualiza un registro de datos según su clave primaria.

PathParam:

- table: nombre del servicio.
- id: lista de valores separados por el campo SEPARATOR uno por cada valor de la clave primaria en el orden en el que se definió en el campo KEYS.

Body: objeto JSON con los datos del registro.

ReturnData: devuelve el mismo objeto JSON que se pasa en el body.

Este servicio permite cambiar la clave primaria. En la lista de id's se identifica la cláusula WHERE mientras que en body se identifica la cláusula SET.

NOTA: aquí la lista de ids puede ser incompleta, lo que permite actualizar elementos que comparten parte de la clave primaria.

Otros servicios adicionales**@GET ("/mapperinfo/:table")**

Devuelve los metadatos del servicio: tipo, nombre, fields, keys,

@GET ("/mapperinfolist")

Devuelve una lista de los servicios desplegados

@GET ("/_inituser")

Devuelve el objeto User de la llamada a UserDao.initUser.

Este método obtiene el login/id del usuario llamando a UserDao.getLogin, y le pasa como parámetro el nombre de la aplicación que tiene desplegado GenericRest.

La llamada a este método invalida la caché y actualiza los atributos/metadatos del usuario sin que tenga que volver a logearse.

@GET ("/_getuser")

Devuelve el objeto User de la llamada a UserDao.getUser.

Este método obtiene el login/id del usuario llamando a UserDao.getLogin, y le pasa como parámetro el nombre de la aplicación que tiene desplegado GenericRest.

Sintaxis de fields

fields -> * | campo | fields ',' fields

campo -> field [':' descripcion] ['#' tipo]

field -> [_0-9A-Za-z]* ['-']

descripcion -> [_0-9A-Za-z]*

tipo -> T ['#' size] | N | F | D | S ['#' sequenceName]

size -> [0-9]*

sequenceName -> [_0-9A-Za-z]*

Fields puede contener únicamente el símbolo '*' para denotar todos los campos resultantes de la ejecución de la consulta, o puede ser una lista parcial de columnas y terminada en '*' para poder especificar diferentes funcionalidades para algunos campos.

Si field acaba en '-' significa que no se tiene en cuenta para el full-text.

Los tipos de datos son:

- T: texto, permite definir el tamaño.
- N: numero
- F, D: Fecha y Date, el primero con tiempo, el segundo sin tiempo, sólo fecha.
- S: secuencia de base de datos, viene acompañada del nombre de la secuencia.

Cuando Estos tipos de datos se mapean según el driver con los diferentes tipos de datos que hay en la base de datos:

- T: CHAR, VARCHAR, VARCHAR2.
- F: DATE, DATETIME, TIMESTAMP.
- N: LONG, LONG RAW, NUMBER, DECIMAL, INT, DOUBLE.

El tipo de dato 'D' es equivalente a 'F' salvo que transforma la entrada y salida de datos al formato de sólo fecha.

Sintaxis de filtro

```
filter -> valor | '['campo']' op valor | '['campo']' op '[' campo ']'
        | '(' filter ')' | filter 'AND' filter | filter 'OR' filter
op -> = | == | < | > | >= | <= | =>
valor -> [_0-9A-Za-z]*
campo -> [#{}._0-9A-Za-z]*
```

Si algunas de las expresiones booleanas contienen 'valor' se entiende que es búsqueda full-text.

'campo' puede hacer referencia:

- al nombre de una columna (ej: IMPORTE)
- un atributo de los metadatos del usuario (ej: #{user.login}, #{user.roles}, #{user.grupos}, #{user.attr.CENTROS})
- un valor de los datos (ej: #{data.SALARIO})
- un valor de la clave primaria (ej: #{id.ID_SOLICITUD})
- el nombre del servicio (ej: #{table})
- un campo de la url (ej: #{ui.CAMPO})
- cualquier referencia tipo Mybatis de los objetos presentes en la llamada.

Resumiendo, los objetos accesibles para 'campo' son: user, table, info, filter, limit, offset, order,

orderby, fields, ui, query, out, id y data.

Sintaxis de SEC_INFO

sec_info -> key '=' valor [',' sec_info]

key -> (S | I | U | D | E)*

valor -> filter [':' valor]

Una línea de sec_info es una lista de pares key/valor para cada una de las operaciones que expone el servicio: S (select), I (insert), U (update), D (delete), E (execute) y la condición que se debe cumplir para que el servicio se ejecute.

La sintaxis de 'valor' es idéntica a la utilizada en el filtro de datos, pero la semántica es diferente:

'valor' sin operación: hace referencia a un rol dentro del array de valores User.getRoles.

'campo' puede hacer referencia:

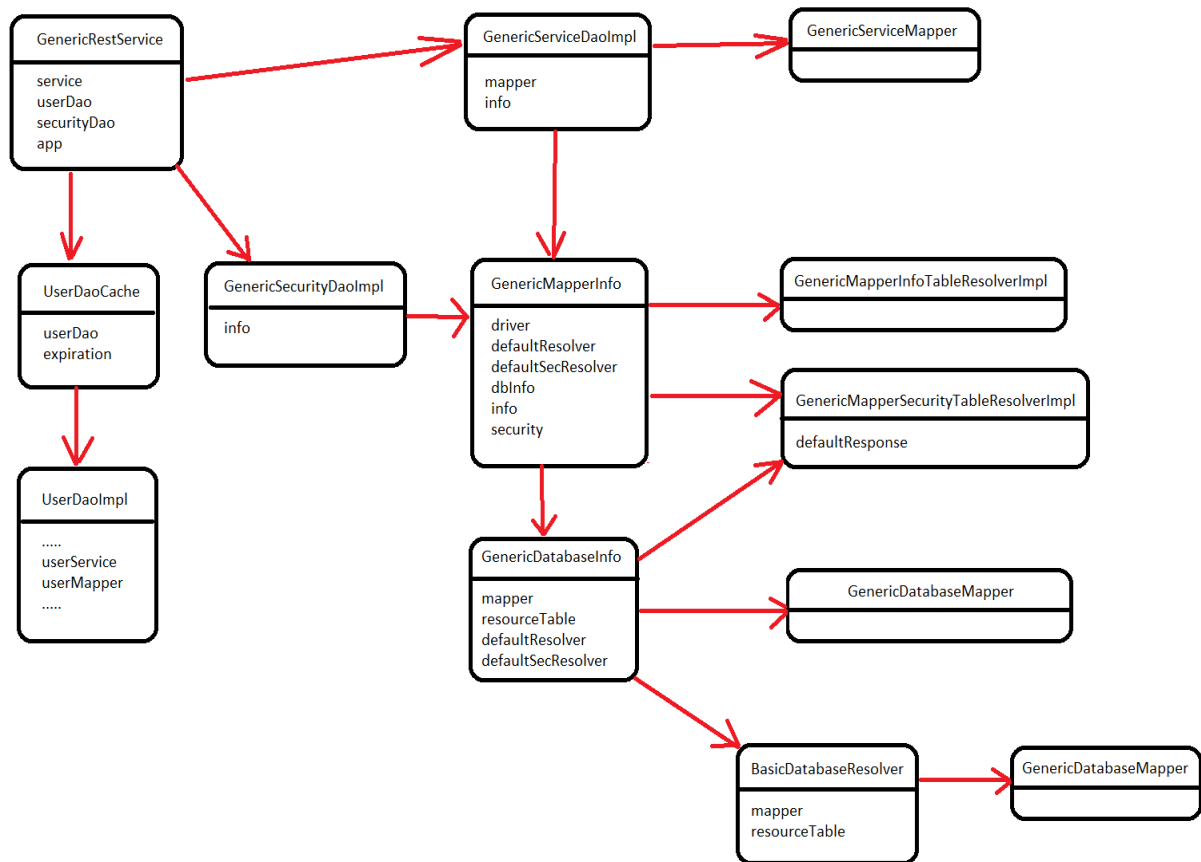
- un atributo de los metadatos del usuario (ej: #{user.login}, #{user.roles}, #{user.grupos}, #{user.attr.CENTROS})
- un valor de los datos (ej: #{data.SALARIO})
- un valor de la clave primaria (ej: #{id.ID_SOLICITUD})
- el nombre del servicio (ej: #{table})

Los objetos accesibles aquí son: user.login, user.roles, user.grupos, user.attr, data, id y table.

Sintaxis de mapperInfo dentro del fichero Spring para info y secInfo

-- TODO --

Estructura de clases



GenericREST implementa un servicio como cualquier otro, tiene asociado un ServiceDao, un ServiceDaoImpl y un ServiceMapper de Ibatis. También tiene asociado un SecurityDao que implementa la seguridad y un UserDao que implementa los atributos/metadatos del usuario.

La implementación del UserDao es específica de cada implantación, su única misión es implementar los métodos "getLogin" (devolver el login/id de la persona que está accediendo al servicio), "initUser" y "getUser" que devuelven los metadatos del usuario. Un usuario es básicamente un contenedor de atributos.

La implementación de SecurityDao corre a cuenta de GenericRest y se apoya en los metadatos del usuario que devuelve UserDao. También se apoya en la información que suministra GenericMapperInfo.

GenericMapperInfo mantiene información de todos los servicios desplegados por GenericRest, su nombre, las columnas, la clave primaria, etc...Esta información proviene de dos fuentes: dentro de la configuración del objeto: "info" y "security" y otro que delega en base de datos la información de los servicios.

La idea de tener dos fuentes es que hay servicios que se pueden definir estáticamente y otros dinámicamente. Un error en los datos podría provocar la no recuperación de la aplicación hasta que se corrijan, esto permitiría tener siempre algún servicio mínimo funcionando.

GenericDatabaseInfo implementa la definición de los servicios en base de datos, tiene asociado el nombre de la tabla que almacenará la definición, y las sentencias SQL asociadas a las diferentes acciones.

Acerca de los Resolvers

Un resolver es un trozo de código que sabe qué sentencia SQL debe hacer en cada momento usando los datos que el servicio REST provee (en función del tipo y de la llamada al resolver le llegan unos datos u otros). Con esa información el resolver es capaz de construir dinámicamente una sentencia SQL usando la misma sintaxis que permite MyBatis en un provider.

En GenericRest hay implementados dos resolvers:

- GenericMapperInfoTableResolverImpl: resuelve el caso más simple del CRUD a una tabla de la base de datos.
- BasicDatabaseResolver: este resolver, además de estar en base de datos, permite indicar la cláusula SELECT, INSERT, DELETE y UPDATE de un CRUD y la de definir el código PL/SQL de la llamada a un FUNCTION o PROCEDURE.

Se pueden crear resolvers particulares con otras funcionalidades, o bien extendiendo de los existentes o implementando la interfaz.

Acerca de los SecurityResolvers

Un SecurityResolver es un trozo de código que decide si una petición puede ejecutarla o no un usuario con todos los parámetros que se le pasan al servicio.

En GenericRest hay implementado un resolver:

- GenericMapperSecurityTableResolverImpl: es el que dota de significado al campo SEC_INFO y define la sintaxis del valor (del par key=valor) y su semántica.

Se pueden crear resolvers que implementen relaciones complejas de seguridad.